Fast Liveness Checking for SSA-Form Programs

Benoit Boissinot  
E.N.S Lyon

Sebastian Hack  
E.N.S Lyon

Daniel Grund  
Saarland University

Fabrice Rastello  
E.N.S. Lyon

Benoît Dupont de Dinechin  
STMicroelectronics

April 7, 2008
Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion
Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion
Why do we need liveness analysis?

Resources analysis
- Scheduling
- Coalescing/Register-allocation
- PRE sensitive to register pressure
## Two approaches

### Classical Approach: Liveness Sets (LS)

For every block boundary, the set of *all* live variables
- Expensive precomputation (space & time), fast query
- Usually, not all computed information is needed
- Adding, (re-)moving instructions ⇒ recompute information

### Our Approach: Liveness Checking (LC)

Answer *on demand*: Is variable live at program point?
- Faster precomputation, slower queries
- Information depends only on CFG and def-use chains
- Information invariant to adding, (re-) moving instructions
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):
- one entry node $r$
- every node reachable from $r$

Definition

$a$ dominates $b$ if every path from the root $r$ to $b$ contains $a$. 
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node $r$
- every node reachable from $r$

Definition

$a$ dominates $b$ if every path from the root $r$ to $b$ contains $a$. 

1 dominates 4?
Dominance

Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):
- one entry node \( r \)
- every node reachable from \( r \)

Definition

\( a \) dominates \( b \) if every path from the root \( r \) to \( b \) contains \( a \).

In the diagram, \( 1 \) dominates \( 4 \).
Dominance

Control Flow Graph
Intermediate representation is a Control Flow Graph (CFG):
- one entry node \( r \)
- every node reachable from \( r \)

Definition
\( a \) dominates \( b \) if every path from the root \( r \) to \( b \) contains \( a \).
Dominance

Control Flow Graph
Intermediate representation is a Control Flow Graph (CFG):
- one entry node \( r \)
- every node reachable from \( r \)

Definition
\( a \) dominates \( b \) if every path from the root \( r \) to \( b \) contains \( a \).
Static Single Assignment (SSA)

Static Single Assignment (SSA) with dominance property:
- (textually) only one definition per variable
- the definition $d$ dominates all its uses $u_i$
Static Single Assignment (SSA) with dominance property:

- (textually) only one definition per variable
- the definition $d$ dominates all its uses $u_i$
Static Single Assignement (SSA) with dominance property:

- (textually) only one definition per variable
- the definition $d$ dominates all its uses $u_i$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$.
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$
Liveness

**Concept**
- Defined in the past: reaching definition
- Used in the future: upward exposed use

**Definition (live-in)**
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

Mathematically,

$$x = 1$$

$$= x$$

$$r = 0$$

1. $r = 0$
2. $x = 1$
3. $= x$
4. $r = 0$
5. $x = 1$
6. $= x$
7. $r = 0$
8. $x = 1$
9. $= x$
**Concept**
- Defined in the past: reaching definition
- Used in the future: upward exposed use

**Definition (live-in)**
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

$$x = 1$$

$$r = 0$$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

![Diagram showing the concept of liveness](image.png)
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

$$x = 1$$

$$r = 0$$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

\[
\begin{align*}
x &= 1 \\
&= 2 \\
&= x \\
r &= 0
\end{align*}
\]
Liveness

**Concept**
- Defined in the past: reaching definition
- Used in the future: upward exposed use

**Definition (live-in)**
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable $a$ is live-in at a node $q$ if there exists a path from $q$ to a node $u$ where $a$ is used and that path does not contain its definition $d$. 

$x = 1 
= x 

x$ is not live at $q$
Liveness

Concept
- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)
A variable \(a\) is live-in at a node \(q\) if there exists a path from \(q\) to a node \(u\) where \(a\) is used and that path does not contain its definition \(d\).
Liveness: precomputation versus queries

- Classical liveness (data-flow):
  - Costly precomputation
  - Almost constant queries

- Our solution:
  - Fast precomputation
  - Queries almost linear in the number of uses
Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion
Goal:
From all the paths from *query* to *use*, remove those going through *def*.

**Highest point**
Last point of the path such that all the following points are below.

If the highest point is dominated by *def* then the whole path is.
Goal:
From all the paths from query to use, remove those going through def.

**Highest point**
Last point of the path such that all the following points are below.

If the highest point is dominated by def then the whole path is.
Goal:
From all the paths from query to use, remove those going through def.

**Highest point**
Last point of the path such that all the following points are below.

If the highest point is dominated by def then the whole path is.
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above def* (not dominated by *def*).

From the remaining *highest points*, test the *descending* reachability to *use*.
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above* def (not dominated by def).

From the remaining *highest points*, test the descending reachability to use.
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

- From this set, remove the points *above* $\text{def}$ (not dominated by $\text{def}$).
- From the remaining highest points, test the descending reachability to $\text{use}$.

*Example 1*
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above def* (not dominated by def).

From the remaining *highest points*, test the *descending reachability to use*.
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above def* (not dominated by def).

From the remaining *highest points*, test the *descending reachability* to *use*.

Example 1
For each node \( q \) of the CFG, compute the set of potential *highest points* of every path starting at \( q \).

From this set, remove the points *above def* (not dominated by *def*).

From the remaining *highest points*, test the *descending reachability* to *use*.
Principle

- For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.
- From this set, remove the points *above def* (not dominated by def).
- From the remaining *highest points*, test the *descending* reachability to *use*.

Example 2
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.

**Example 2**

```
  r=0
     /   \
   def 9
     |     |  use 7
     |     |    q
     |     /     \
 1 5 6 8
  |   |   |
2 3 4
```

*Example 2 diagram*
Principle

- For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.
- From this set, remove the points *above* def (not dominated by def).
- From the remaining *highest points*, test the descending reachability to use.

Example 2

```
   r=0
   \downarrow
  def
  \downarrow
  1  \downarrow
  \downarrow
  9

  \downarrow
  5

  \downarrow
  6

  \downarrow
  \downarrow
  2 \downarrow
  \downarrow
  8

  \downarrow
  \downarrow
  3 \downarrow
  \downarrow
  4

  \downarrow
  \downarrow
  7

  \downarrow
  \downarrow
  \downarrow
  \downarrow
  \downarrow
  q
```
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above* $\text{def}$ (not dominated by $\text{def}$).

From the remaining *highest points*, test the *descending* reachability to $\text{use}$.
For each node $q$ of the CFG, compute the set of potential *highest points* of every path starting at $q$.

From this set, remove the points *above* def (not dominated by def).

From the remaining *highest points*, test the *descending* reachability to use.
For each node $q$ of the CFG, compute the set of potential highest points of every path starting at $q$.

From this set, remove the points above $\text{def}$ (not dominated by $\text{def}$).

From the remaining highest points, test the descending reachability to $\text{use}$.
Algorithm

Precomputation

1. Compute transitive closure on the reduced graph $G'$
   - $G' = \text{CFG without DFS back edges (cycle-free)}$
   - Simple to compute: post-order traversal
2. For each node $q$ compute a set $T_q$ of possible highest points (back-edge targets)
   - Also simple to compute: pre-order and post-order traversal

Query

- For each use:
  - For each $t \in T_q$ dominated by def:
    - Test reachability in the reduced graph
Implementation Tricks

- Reachability and $T_q$ can be efficiently implemented as bitsets.
- For reducible CFGs there is exactly one “highest” back-edge target:
  - dominates all the other back-edge targets
  - sufficient to check from there
- Hence, order nodes according to dominance:
  - “highest” node is first set bit in $T_q$
Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion
Evaluation

Setup

- Implemented in LAO, code generator developed by STMicroelectronics
- Benchmarked with a subset of SPEC2000 (CINT)
- Liveness-analysis used during SSA deconstruction

The main factors influencing the speed of our algorithm are:

- the number of uses per variable ($\#_{\text{uses}}$)
- the number of basic blocks ($\#_{\text{BB}}$)
- the number of CFG edges ($\#_{\text{edges}}$)
# Quantitative Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maximum</th>
<th>% ≤ 1</th>
<th>% ≤ 2</th>
<th>% ≤ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>51</td>
<td>65.64</td>
<td>86.38</td>
<td>92.81</td>
</tr>
<tr>
<td>175.vpr</td>
<td>75</td>
<td>70.36</td>
<td>88.90</td>
<td>93.93</td>
</tr>
<tr>
<td>176.gcc</td>
<td>422</td>
<td>73.99</td>
<td>87.81</td>
<td>92.42</td>
</tr>
<tr>
<td>181.mcf</td>
<td>46</td>
<td>66.91</td>
<td>83.50</td>
<td>89.33</td>
</tr>
<tr>
<td>186.crafty</td>
<td>620</td>
<td>72.98</td>
<td>90.09</td>
<td>93.85</td>
</tr>
<tr>
<td>197.parser</td>
<td>96</td>
<td>65.12</td>
<td>86.75</td>
<td>94.26</td>
</tr>
<tr>
<td>254.gap</td>
<td>156</td>
<td>70.46</td>
<td>85.95</td>
<td>91.26</td>
</tr>
<tr>
<td>255.vortex</td>
<td>254</td>
<td>65.99</td>
<td>90.80</td>
<td>95.02</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>36</td>
<td>69.89</td>
<td>89.89</td>
<td>94.47</td>
</tr>
<tr>
<td>300.twolf</td>
<td>165</td>
<td>69.71</td>
<td>87.59</td>
<td>93.23</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>620</strong></td>
<td><strong>71.30</strong></td>
<td><strong>87.85</strong></td>
<td><strong>92.76</strong></td>
</tr>
</tbody>
</table>
## Quantitative Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average</th>
<th>% ≤ 32</th>
<th>% ≤ 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>33.35</td>
<td>69.51</td>
<td>85.36</td>
</tr>
<tr>
<td>175.vpr</td>
<td>34.45</td>
<td>68.88</td>
<td>84.44</td>
</tr>
<tr>
<td>176.gcc</td>
<td>38.96</td>
<td>72.85</td>
<td>86.03</td>
</tr>
<tr>
<td>181.mcf</td>
<td>20.31</td>
<td>84.61</td>
<td>100.00</td>
</tr>
<tr>
<td>186.crafty</td>
<td>69.28</td>
<td>59.63</td>
<td>76.14</td>
</tr>
<tr>
<td>197.parser</td>
<td>23.60</td>
<td>84.82</td>
<td>93.49</td>
</tr>
<tr>
<td>254.gap</td>
<td>32.89</td>
<td>67.60</td>
<td>87.44</td>
</tr>
<tr>
<td>255.vortex</td>
<td>26.46</td>
<td>77.57</td>
<td>90.68</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>22.97</td>
<td>78.37</td>
<td>91.89</td>
</tr>
<tr>
<td>300.twolf</td>
<td>56.97</td>
<td>59.47</td>
<td>77.36</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>35.21</strong></td>
<td><strong>72.71</strong></td>
<td><strong>87.18</strong></td>
</tr>
</tbody>
</table>
## Runtime Experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Precomputation</th>
<th>Queries</th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3.12</td>
<td>0.53</td>
<td>1.16</td>
</tr>
<tr>
<td>175.vpr</td>
<td>2.17</td>
<td>0.48</td>
<td>1.41</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3.03</td>
<td>0.26</td>
<td>1.00</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.85</td>
<td>0.44</td>
<td>1.39</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2.78</td>
<td>0.49</td>
<td>0.73</td>
</tr>
<tr>
<td>197.parser</td>
<td>2.13</td>
<td>0.49</td>
<td>1.54</td>
</tr>
<tr>
<td>254.gap</td>
<td>3.45</td>
<td>0.52</td>
<td>2.08</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1.67</td>
<td>0.45</td>
<td>1.32</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>3.45</td>
<td>0.51</td>
<td>2.32</td>
</tr>
<tr>
<td>300.twolf</td>
<td>4.76</td>
<td>0.49</td>
<td>1.92</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2.94</strong></td>
<td><strong>0.36</strong></td>
<td><strong>1.16</strong></td>
</tr>
</tbody>
</table>
Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion
Contributions

- Novel approach for liveness checking relying only on the CFG
- Fast construction algorithm
- Overall speedup in most cases
Future Work

- Dynamic update for CFG transformations
- Memory efficient reachability
- Use information available from the loop nesting forest
The End

Thank you!